Using a Filter-Based SQP Algorithm in a Parallel Environment

Gerhard Venter (gventer@sun.ac.za)*

University of Stellenbosch, Stellenbosch, South Africa

Garret N. Vanderplaats (vanderplaats@vrand.com)†

Vanderplaats Research and Development, Inc., Monterey, CA 93940

A parallel, filter-based, sequential quadratic programming (SQP) algorithm is implemented and tested for typical general-purpose engineering applications. Constrained engineering test problems, including a finite element simulation, with up to 512 design variables are considered. The accuracy and serial performance of the filter-based algorithm are compared against that of a standard SQP algorithm. The parallel performance of the algorithm is evaluated, using up to 52 cores on a Linux Cluster. The results indicate that the filter-based algorithm competes favorably with a standard SQP algorithm in a serial environment. However, the filter-based algorithm exhibits much better parallel efficiency due to the lack of a one dimensional search.

I. Introduction

GENERAL-purpose optimization is a powerful engineering tool that allows designers to couple an optimization algorithm with an analysis tool. Several commercially available tools exist that aid the designer in this process. These tools typically provide a number of robust optimization algorithms and a process integration tool that simplifies the coupling process. Examples include VisualDOC from Vanderplaats Research and Development, iSIGHT from Dassault Systèmes, modeFRONTIER from Esteco and Optimus from

*Professor, Department of Mechanical and Mechatronic Engineering, Stellenbosch, South Africa, AIAA Senior Member

†Chief Executive Officer, Vanderplaats Research and Development, Inc., 126 Bonifacio Place, Suite F, Monterey, CA 93940, AIAA Fellow

Noesis Solutions. Despite the availability of these tools, and the power that the underlying technology brings to the design process, general-purpose optimization is not being accepted in industry at the rate one would expect. Some of the reasons traditionally provided for this lack of acceptance are that optimization is difficult to use, the coupling of the optimizer to the analysis code is painful and that an optimization study can take a long time to complete. The mentioned tools adequately address the first two issues. However, the third issue is more problematic and is the focus of the current paper.

By its very nature, a single optimization run requires multiple analyses to complete. A typical optimization study can easily require one or two orders of magnitude more time than a single analysis. An obvious way to deal with this problem is to exploit the advantages offered by parallel computing. Most of the commercially available general-purpose optimization tools provide some parallel capabilities, but these are often difficult to setup and use, and more importantly do not fully exploit the advantages available from parallel processing. Although parallel processors are becoming more easily available to designers, think for example of multi-core processors that are becoming common place in personal computers, there are few effective parallel optimization algorithms available.

When implementing a parallel optimization algorithm, one of two approaches can be followed. The first is to devise a parallel implementation of an existing algorithm, the second is to develop a brand new algorithm, specifically designed for a parallel environment. The advantage of using the first approach is that one can leverage existing technology and build on algorithms that are known to be robust, efficient and applicable to a wide range of problems. The problem with extending existing algorithms that were explicitly developed for a serial environment to a parallel environment, is that the most efficient serial algorithm does not necessarily provide the most efficient parallel algorithm (e.g., Venter and Watson[1]). The advantage of the second approach is that one can potentially gain more efficiency from the parallel environment. However, developing a robust optimization algorithm that is applicable to a wide range of problems is not a trivial task.

Many researchers that look at parallelizing existing algorithms concentrate on zero-order algorithms like Genetic Algorithms (GAs) or Particle Swarm Optimization (PSO), with varying success. For example, one can obtain near theoretical speedup by implementing an asynchronous PSO algorithm (e.g., Venter and Sobieszczanski-Sobieski[2] and Koh et al.[3]). Even so, these zero-order algorithms are computationally inefficient, need parameter tuning that is problem dependent and are typically limited to problems with smaller numbers of design variables. Even within a parallel environment, these algorithms rarely compete with the efficiency of gradient-based algorithms. As a result, in industry, most engineering optimization studies are still performed using gradient-based optimization.

Parallel processing is becoming more readily available to designers, especially in the

form of a small number of processing units. Within this environment, the authors believe that fully exploiting the benefits provided by an efficient and robust gradient-based algorithm is an important enabling technology for the use of general-purpose optimization in industry. The current paper is guided by the basic observations from Venter and Watson,[1] which implemented parallel versions of three widely used gradient-based algorithms namely, sequential quadratic programming (SQP), the modified method of feasible directions (MMFD) and sequential linear programming (SLP). The commercially available Design Optimization Tools[4] (DOT) implementation of these algorithms was used for the study. For the example problem considered, the results indicated that in a serial environment the MMFD algorithm was the most efficient and the SLP algorithm was the least efficient. However, in a parallel environment the SLP algorithm required the smallest elapsed time to complete. Even though the SLP algorithm required more function evaluations to complete, it could more efficiently exploit the parallel environment, since it required no one-dimensional search. Although the SLP algorithm performed best for the example problem considered, it is in general not considered a good algorithm in the class of the SQP and MMFD algorithms and in many cases may not converge to the optimum solution.

The current paper will concentrate on a parallel implementation of a robust, efficient and widely used gradient-based algorithm, namely the sequential quadratic programming (SQP) algorithm (e.g., Wilson,[5] Han[6] and Powell[7]). The application region is general-purpose optimization in engineering with problems that have anywhere from one or two, to several hundred design variables. Most general-purpose optimization problems in engineering fall in this category, since the number of design variables is limited by the fact that gradient information is typically not available and must be calculated using finite difference gradient calculations. The current paper will build on the lessons learned from Venter and Watson[1] to investigate how an efficient SQP algorithm can be implemented in parallel.

II. Gradient-based Optimization Background

When parallelizing an algorithm, one typically starts by identifying the main sources of computational cost. For most gradient-based optimization algorithms, one can naturally divide the computational cost into three components. First is the computational cost associated with obtaining the required gradient information, typically in the form of finite difference gradient calculations. Second is the cost of performing the one-dimensional search, typically in the form of a polynomial approximation or a Golden Section search. Third is the cost associated with the optimization algorithm itself, for example solving the

direction finding sub-problem. If the analysis code requires any significant time to complete, even as little as just a few seconds, the time required to complete the finite difference gradient calculations typically dominates the total time. This is followed by the time required to complete the one-dimensional search calculations, while the time associated with the algorithm itself is typically negligible. In most cases, engineering simulations take enough time that one can ignore the time required to perform the computations associated with the algorithm. When this is not the case, for example when the analysis takes only a fraction of a second to complete, the overall time required to complete the optimization study is typically small and parallelization is most probably not important to start with. The present work will concentrate on cases where the analysis time takes anywhere from a few seconds to a few minutes or longer per analysis.

Since the finite difference gradient calculations tend to dominate the overall time for most engineering problems and are easily parallelized, performing these computations in parallel has been the focus of many previous efforts to parallelize existing gradient-based algorithms (e.g., Venter and Watson[1]). The one-dimensional search computations, although having a significant contribution to the overall computational burden, is inherently a serial process that is difficult to parallelize. As a result many parallel implementations of existing algorithms still perform these computations in a serial fashion. The present paper will look at ways to overcome this problem for the SQP algorithm, by selecting a variant of the traditional SQP algorithm that does not depend on a one-dimensional search.

The general non-linear optimization problem that will be considered in the present paper, can be summarized as follows:

$$\begin{aligned} \text{Minimize:} \quad & f(\boldsymbol{x}) \\ \text{Such That:} \quad & g_j(\boldsymbol{x}) \leq 0 \qquad j = 1, m \\ & x_i^l \leq x_i \leq x_i^u \quad i = 1, n \end{aligned} \tag{1}$$

where f is the objective function, g_j are inequality constraints and $\boldsymbol{x}$ is a vector of n design variables. x_i^u and x_i^l represent the upper and lower bounds on the design variables, referred to as side constraints.

III. The Traditional SQP Algorithm

Sequential Quadratic Programming (SQP) is a popular general-purpose optimization algorithm that is widely used to solve problems of the type summarized in Eq. (1). The SQP algorithm creates an approximate quadratic programming (QP) sub-problem that is used to find a search direction. The QP subproblem is obtained by creating a quadratic

approximation of the objective function and linear approximations of the constraints as shown in Eq. (2) (e.g., Powell[7])

$$\begin{aligned} \text{Minimize:} \quad & Q(\boldsymbol{x}) = F(\boldsymbol{x}) + \nabla F(\boldsymbol{x})^T \boldsymbol{s} + \frac{1}{2}\boldsymbol{s}^T \boldsymbol{B}\boldsymbol{s} \\ \text{Such That:} \quad & g_j(\boldsymbol{x}) + \nabla g_j(\boldsymbol{x})^T \boldsymbol{s} \leq 0 \qquad j = 1, m \end{aligned} \tag{2}$$

where Q is the quadratic objective function, $\boldsymbol{B}$ is a positive definite matrix which is initially the identify matrix and $\boldsymbol{s}$ is the unknown search direction. The $\boldsymbol{B}$ matrix is updated after each iteration to approximate the Hessian of the Lagranian, using the the standard BFGS update scheme (e.g., Powell[7]). The design variables are the unknown components of the n^{th} dimensional search direction, $\boldsymbol{s}$. The QP subproblem can be solved with a specialized QP algorithm, but it is a well posed optimization problem that can also be solved with any other non-linear optimization algorithm like MMFD.

It is well known that if the SQP algorithm is started far from a solution, it may not converge to a local optimum if the search direction obtained from Eq. (2) is used directly. To ensure convergence to a local optimum, a one-dimensional search is typically performed to obtain a step size α. A new design point is then obtained from the search direction and the step size, as follows:

$$\boldsymbol{x}^q = \boldsymbol{x}^{q-1} + \alpha \boldsymbol{s}^q \tag{3}$$

A new design point is only accepted if a specified merit function is reduced during the one-dimensional search. Most often, the merit function is specified as an exterior penalty function that combines the objective function and a measure of the constraint violations. A reduction in the merit function thus implies a reduction in the objective function and/or constraint violation. A popular merit function, based on the Lagranian function, (e.g., Powell[7] as outlined in Vanderplaats[8]) is shown in Eq. (4), where an initial value of $\alpha = 1$ is typically a good starting value.

$$\begin{aligned} \text{Minimize:} \quad & \phi(\boldsymbol{x}) = F(\boldsymbol{x}) + \sum_{j=1}^{m} u_j \left\{ max[0, g_j(\boldsymbol{x})] \right\} \\ \text{where:} \quad & \boldsymbol{x} = \boldsymbol{x}^{q-1} + \alpha \boldsymbol{s} \\ & u_j = |\lambda_j| \qquad \text{First iteration} \\ & u_j = max\left[|\lambda_j|, \frac{1}{2}\left(u'_j + |\lambda_j| \right) \right] \qquad \text{Subsequent iterations} \end{aligned} \tag{4}$$

In Eq. (4) $u'_j = u_j$ from the previous iteration and λ_j refers to the Lagrange multipliers obtained from the QP problem summarized in Eq. (2).

With the optimum step size $\alpha^\star$ obtained from solving Eq. 4, one can update the current design point using Eq. (3), update the $\boldsymbol{B}$ matrix using the BFGS scheme (e.g., Powell[7]) and repeat the process until convergence.

A. Parallel Implications

In general the SQP algorithm is thus a two step process, where one first determines the search direction (which depends on 1^{st} order gradient information) and second determines the step size, the one dimensional search. This two step process is problematic from a parallelization point of view, and is discussed in more detail here.

In general, the time required to perform an optimization study, using a two step gradient-based algorithm like the SQP algorithm, can be summarized as

$$T_{total} = nIter\,(T_{grad} + T_{1D}) + T_{int} \tag{5}$$

where T_{total} refers to the total time required to complete the optimization study, $nIter$ the number of design iterations required to achieve convergence, T_{grad} the time required to perform a single set of gradient calculations, T_{1D} the time required to perform a single one-dimensional search and T_{int} the time required by the optimization algorithm itself.

To investigate the parallel speedup that can be achieved from this general setup, consider the following assumptions: (1) the T_{int} component is small and can be neglected; (2) each analysis takes the same time denoted by T_{anal}; (3) the total time is normalized to obtain $\overline{T}_{total} = T_{total}/T_{anal}$; (4) enough processors are available and perfect speedup is obtained to give $T_{grad}/T_{anal} = 1$; and (5) the one-dimensional search is performed in series. Experience with the commercially available SQP algorithm provided as part of the Design Optimization Tools (DOT) library,[4] shows that the one-dimensional search (using a polynomial approximation) on average takes about three analyses, yielding $T_{1D}/T_{anal} = 3$. The total time to complete the optimization study in our assumed, best case parallel environment then becomes:

$$\overline{T}_{total} = nIter\,(1 + 3) \tag{6}$$

Equation (6) can be further reduced to Eq. (7) if the finite difference gradients are calculated at each point visited during the one-dimensional search.

$$\overline{T}_{total} = 3\,nIter \tag{7}$$

Ideally, it would be best to also parallelize the one dimensional search, which will reduce Eq. (6) to:

$$\overline{T}_{total} = nIter\,(1+1) \tag{8}$$

This is half the time of only parallelizing the finite difference calculations alone (Eq. (6)) and two thirds the time of calculating the finite difference gradients at each point visited during the one-dimensional search (Eq. (7)). However if the gradient calculations and the one-dimensional search can be combined into a single operation, the overall time becomes:

$$\overline{T}_{total} = nIter \tag{9}$$

Equation (9) can be achieved in one of two ways: (1) by performing the one-dimensional search in parallel and by calculating the gradient information at each point used in the parallel one-dimensional search or (2) by using an algorithm that has no one-dimensional search. Approach (1) is problematic, since it is difficult to parallelize the one-dimensional search effectively. One approach may be to evenly divide the one-dimensional search domain into k pieces and perform all k evaluations in parallel. However, to get the performance outlined in Eq. (9) one would also need to perform n (where n is the number of design variables) additional evaluations at each of the k points, for a total of $k\,(n+1)$ analyses per iteration. To put this in perspective, from Eq. (5) the standard serial implementation would require $n+3$ analyses per iteration. For a problem with 50 design variables and using $k = 20$ the serial approach would require 53 analyses per iteration, while Approach (1) would require 1100 analyses per iteration. In addition, Approach (1) will only work if the upper bound of the one-dimensional search domain is known, which is generally not the case. In contrast, Approach (2) would require only $n+1$ analyses per iteration. For the above example this is 51 analyses per iteration. Approach (1) may thus be a valid approach for future implementations where larger numbers of processors (more than a 100 or so) are readily available, while Approach (2) may be more appropriate for the current environment where most designers have access to smaller numbers of processors (4 or 8).

The next section will look at a variant of the traditional SQP algorithm that provides similar performance, without the use of a one-dimensional search.

IV. Filter based SQP

Fletcher and Leyffer[9] introduced a variant to the traditional SQP algorithm that does not use a penalty function to guarantee convergence to a local optimum. Their goal was to introduce an algorithm that is easy to implement, robust and that converges rapidly to a

local optimum. From a parallel perspective, the attractive feature of their algorithm is that no penalty function, and thus no one-dimensional search, is required. It is thus possible to use a well-established, robust, algorithm efficiently in a parallel environment.

The algorithm by Fletcher and Leyffer alleviates some implementation issues associated with the traditional SQP algorithm. First, it is no longer necessary to deal with the problem dependent penalty function and associated penalty parameters. Second, when a penalty function is used, the super-linear convergence associated with the SQP algorithm can be destroyed. This is known as the Maratos effect.[10] For super-linear convergence, $\alpha^\star$ needs to be close to unity in the final iterations before convergence. However, this is often prevented by the penalty function that returns an $\alpha^\star$ value much less than one. The new algorithm also does not suffer from the Maratos effect.

Fletcher and Leyffer[9] proposed the use of a bi-objective formulation to eliminate the need of a penalty function. Their approach is based on the observation that there are two competing aims in non-linear programming. The first is to minimize the objective function f and the second is to minimize the constraint violation. The traditional SQP algorithm makes use of a penalty function approach to combine these two objective functions into a single objective minimization problem. Fletcher and Leyffer replaces the penalty function with a bi-objective formulation as follows

$$\begin{aligned} &\text{Minimize:} \quad f(\boldsymbol{x}) \\ &\text{Minimize:} \quad h\left(g(\boldsymbol{x})\right) \end{aligned} \tag{10}$$

where the constraint satisfaction $h(g(\boldsymbol{x}))$ is expressed as:

$$h\left(g(\boldsymbol{x})\right) = \sum_{j=1}^{m} max\left(0, g_j(\boldsymbol{x})\right) \tag{11}$$

Equation (10) is solved using a filter approach. In this context, the filter is simply the set of non-dominated solutions found so far, where a solution (f^k, h^k) dominates another solution (f^l, h^l) if and only if both $f^k \leq f^l$ and $h^k \leq h^l$. The filter thus provides the most current state of the Pareto front, and consists of (f^i, h^i) point pairs. The points in the filter is constantly updated as new solutions are generated. The filter is used to decide if a newly generated candidate solution should be accepted or rejected. The filter is thus a direct replacement of the penalty function (see Eq. (4)) as a merit function to determine if a new SQP solution has made enough progress towards the optimum. Using the filter, a candidate solution is only accepted if it is not dominated by any other point currently in the filter. Only solutions that would thus provide a new point on the Pareto front is accepted by the filter. The filter approach is implemented by first obtaining a candidate

SQP solution. This candidate solution is then compared to all points currently in the filter. If it is not dominated by any point currently in the filter, the candidate solution is accepted by the filter as a valid new SQP point and is also included into the filter. Whenever a new solution is included into the filter, the filter is maintained by removing any existing points that is dominated by this new solution. Implementation specific details are discussed below, followed by a detailed outline of the algorithm provided in Algorithm 1.

In addition to the filter, a trust-region or move limit strategy is implemented to guarantee convergence to a local optimum. The trust-region simply limits the magnitude of the search direction and is implemented by adding a constraint to Eq. (2) as shown in Eq. (12).

$$\begin{aligned} \text{Minimize:} \quad & Q(\boldsymbol{s}) = F(\boldsymbol{x}) + \nabla F(\boldsymbol{x})^T \boldsymbol{s} + \frac{1}{2}\boldsymbol{s}^T \boldsymbol{B}\boldsymbol{s} \\ \text{Such That:} \quad & g_j(\boldsymbol{x}) + \nabla g_j(\boldsymbol{x})^T \boldsymbol{s} \leq 0 \qquad j = 1, m \\ & \|\boldsymbol{s}\|_\infty \leq \rho \end{aligned} \tag{12}$$

When a new solution is rejected by the filter, the trust region radius ρ is reduced and the SQP step is repeated. As with any trust region approach, reducing the trust region radius can lead to a situation where no feasible solution exists within the trust region. This will lead to an infeasible QP solution and calls for a restoration phase. Fletcher and Leyffer suggests a simple restoration phase that consists of minimizing $h(g(\boldsymbol{x}))$ until a feasible solution is found. In the present work the design variables are normalized and an initial trust-region radius of 0.5 is used. If the new point is accepted by the filter and the maximum search direction component is equal to the trust-region radius, the radius is increased by a factor of two. If the new point is rejected by the filter, the radius is reduced by a factor of four. The filter-based SQP algorithm can then be outlined as shown in Algorithm 1 below.

Only the basic algorithm, as outlined above, was implemented here. Fletcher and Leyffer proposed several heuristics to improve both the robustness and the efficiency of the basic algorithm. According to Fletcher and Leyffer these heuristics have only a small impact on the performance of the algorithm and was thus not investigated here. The code implemented and presented here was thoroughly checked for correctness and does present representative results for the filter-based SQP algorithm. However, the code should be classified as research quality at best, with the main focus of illustrating the concepts rather than fine tuning performance. Further enhancements in terms of robustness and efficiency can be expected and the results presented here can thus be considered as a lower bound on the true potential of the algorithm. Especially, when comparing against an established optimizer like DOT, that has been used commercially for more than 30 years.

Algorithm 1 Filter based SQP algorithm

Given $\boldsymbol{x}^0$, ρ and $k = 0$

repeat
  Solve the QP (Eq. 12) to obtain $\boldsymbol{s}^k$
  if QP is infeasible **then**
    Find a new point $\boldsymbol{x}^{k+1}$ in the restoration phase
  else
    Set $\boldsymbol{x}^{k+1} \Leftarrow \boldsymbol{x}^k + \boldsymbol{s}^k$
    if (f^{k+1}, h^{k+1}) is accepted by the filter **then**
      Accept $\boldsymbol{x}^{k+1}$
      Add (f^{k+1}, h^{k+1}) to the filter
      Possibly increase the trust region radius ρ
    else
      Reject step and set $\boldsymbol{x}^{k+1} \Leftarrow \boldsymbol{x}^k$
      Reduce the trust region radius ρ
    end if
  end if
  set $k \Leftarrow k + 1$
until convergence

V. Parallel Implementation and Testbed

The advantage of using the filter-based SQP in a parallel environment is that the penalty approach is not used and as a result the one-dimensional search is eliminated. In a parallel environment, the filter-based algorithm thus effectively reduces the traditional SQP algorithm from a two step to a single step process where all the analyses within a design iteration can be performed all at once. Recall from Section III that the time spend in the optimization code was neglected for general-purpose engineering optimization applications. This is a reasonable assumption when the analysis time of a single analysis becomes significant. Using this assumption, an initial evaluation of the two algorithms can be done in a serial environment based on the number of iterations, and the number of steps per iteration, required for convergence. If the traditional and the filter-based algorithms both require the same number of iterations for convergence, the filter-based algorithm will be roughly twice as fast in an ideal parallel environment, as discussed in Section III and outlined in Eqs. (6) and (9). Alternatively, as long as the filter-based algorithm require less than twice the number of iterations as compared to the traditional SQP, the filter-based approach would still be more efficient in a parallel environment.

The parallel implementation of the filter-based algorithm is fairly straight-forward. The only function evaluations associated with each design iteration is the finite difference gradient calculations and these can easily be performed in parallel. The parallel implemen-

tation used here, makes use of the OpenMPI[11] implementation of the Message Passing Interface (MPI) standard. OpenMPI was used to create a master-worker parallel algorithm. In this implementation, the master processor decides what work should be done and then assigns a task to each worker processor. As soon as a worker processor is done with its task it reports back to the master processor and checks if there are more tasks available. This master-worker configuration thus provides dynamic load balancing within the parallel environment.

It is important to note that a synchronous parallel implementation is used, where all analyses within a given design iteration is completed before analyses from the next iteration is started. The synchronous implementation is a direct result of using a gradient-based algorithm and is a good example of a drawback associated with parallelizing an existing serial algorithm. The synchronous implementation is required, since a new search direction can only be calculated once all the gradient information is available. In some cases, a synchronous implementation can lead to situations where all the worker processors are waiting for a single analysis to complete, thus resulting in poor parallel speedup. This can easily happen when: (1) the time required to complete a single analysis depends on the design point being analyzed; (2) a heterogeneous system of processing units is used; and (3) the number of analyses is not an integer multiple of the number of processors. Unfortunately, gradient based optimization and in particular the SQP algorithm does not provide for an asynchronous implementation, where analyses from the next design iteration can be analyzed in the current design iteration. The fact that a synchronous implementation is used, means that a drop-off in parallel efficiency is expected as the number of processors is increased, since more processors will be idle at the end of each design iteration.

The parallel code was tested on the high performance computing facility at Stellenbosch University. This facility consists of a Linux cluster that has a total of 168 2.83 GHz Xeon cores (21 compute nodes, each with 2 quad core processors), 2 GByte memory per core and 300 GByte local disk storage for each compute node. The parallel benchmarking were completed on a 64 core Linux cluster made available by the High Performance Computing Center Stuttgart (HLRS) of the University of Stuttgart. This cluster has a similar setup than the Stellenbosch cluster, consiting of 8 compute nodes, each with 2 quad core 3 Ghz Xeon processors, 0.5 GByte memory per core and a 4 GByte RAM disk for local storage.

VI. Numerical Results

Engineering example problems were selected to illustrate the parallel speedup that can be achieved from the proposed implementation. The first step was to compare the performance of the filter-based SQP algorithm with that of a traditional algorithm in a

serial environment. An analytical example problem, where each analysis is completed in a small fraction of a second is used for this purpose. The goal here was to show that the filter based algorithm provides roughly the same performance and accuracy as a traditional SQP algorithm. The traditional SQP implementation that was selected for the comparison, is the commercially available SQP algorithm provided as part of the Design Optimization Tools (DOT) library.[4]

The second step was to evaluate the parallel performance of the filter-based algorithm, using a more realistic example problem. The second example problem makes use of a numerical simulation, which is representative of a general-purpose engineering application. Although the analysis time is still pretty small (it took about 50 seconds to complete a single analysis), the number of design variables considered is representative of typically engineering applications. In this case, the performance of the filter based algorithm is compared against a parallel implementation of the traditional SQP algorithm provided in DOT. The parallel implementation of the DOT algorithm is similar to that described in Venter and Watson,[1] where a master-worker paradigm is used to perform the finite difference gradient calculations in parallel.

Although example problems that require relatively small computational effort are considered in both cases, the number of design variables (maximum of 512 for problem 1 and 50 for problem 2) is representative of medium to large general-purpose engineering problems using finite-difference gradient calculations.

A. Example problem 1: Cantilevered Beam

Before considering the parallel performance of the new algorithm, it was necessary to first investigate the accuracy of the filter-based algorithm compared to that of the traditional algorithm. This comparison was conducted in a serial environment by solving a single example problem for different number of design variables. The cantilevered beam problem from Vanderplaats[8] was considered as an example problem, as shown in Fig. 1. In the current paper, a beam with a fixed length of 0.5 m, an applied load of 50 kN and a material with Young's Modulus of 200 GPa was used. The beam is divided into segments of equal length, each with width b_i and height h_i.

The optimization problem is defined as changing the width and height of each segment to minimize the volume, while satisfying displacement, stress and geometric constraints as shown in Eq. (13).

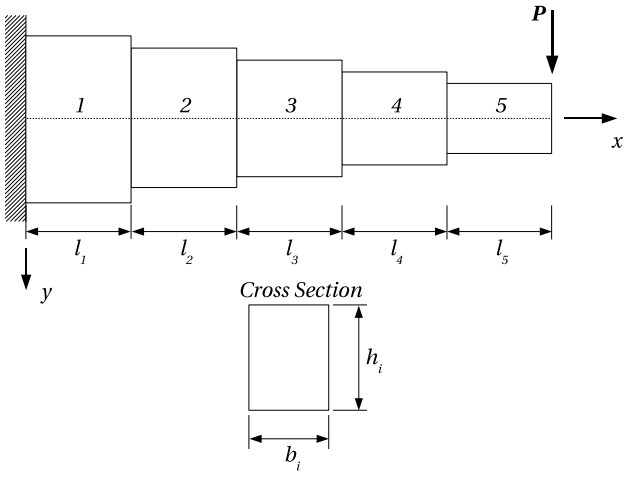


Figure 1. Beam example

$$
\begin{aligned}
\text{Minimize:} \quad & Volume = \sum_{i=1}^{nSeg} b_i\, h_i\, l_i \\
\text{Such That:} \quad & \frac{\sigma_i}{\overline{\sigma}} - 1 \le 0 \qquad\qquad i = 1, nSeg \\
& h_i - 20\, b_i \le 0 \qquad\qquad i = 1, nSeg \\
& \frac{y_n}{\overline{y}} - 1 \le 0 \\
& b_i \ge 1.0 \qquad h_i \ge 5.0
\end{aligned}
\tag{13}
$$

The stress and geometric constraints are applied to each segment, while only the tip displacement is constrained. The maximum stress $\overline{\sigma}$ is equal to 140 MPa, while the maximum displacement $\overline{y}$ is equal to 25 mm.

The problem is implemented in such a way that the number of segments ($nSeg$) can easily be changed to control the problem size. The number of design variables was varied from 2 to 512 to cover the application range of small to large general-purpose engineering optimization problems. To account for the influence of the starting point on the performance of the algorithms, 10 random starting points were generated for each number of design variables considered. Both algorithms were started from the same random starting points with the mean and standard deviation of the final objective function value and number of iterations required for convergence summarized in Table 1.

Both algorithms were able to consistently find good optimum solutions. The maximum difference in objective function value between the two algorithms for a given number of design variables was only 1.72%. Both algorithms found slightly worse optimum solutions with an increase in the number of design variables. Tightening of the convergence criteria should solve this problem. For the study conducted here the same convergence criteria

Table 1. Serial accuracy and performance comparison

Design Variables	Mean Objective		DOT Iterations		Filter Iterations	
	DOT	Filter	Mean	Std Dev	Mean	Std Dev
2	89526.64	89265.34	7.50	2.33	6.70	1.49
4	72967.67	72822.04	7.80	1.66	7.30	1.42
8	66157.05	66131.59	10.20	1.99	11.30	2.76
16	64503.92	64503.78	17.30	1.49	25.80	5.62
32	63930.08	63923.66	21.30	4.84	27.30	4.34
64	64093.46	63873.79	14.80	5.23	24.30	6.23
128	64044.38	63832.60	12.70	2.00	22.70	5.95
192	64761.40	64199.49	12.40	3.93	21.00	5.98
256	65266.88	64233.99	13.00	2.45	21.60	5.46
320	65780.66	65255.38	17.30	8.16	18.40	4.00
384	65065.31	64824.64	19.90	11.89	28.10	11.94
448	65385.96	65354.71	17.70	8.80	29.60	12.51
512	65321.25	66442.47	23.00	10.45	20.90	6.24

were used for both algorithms and were kept constant for all variations considered.

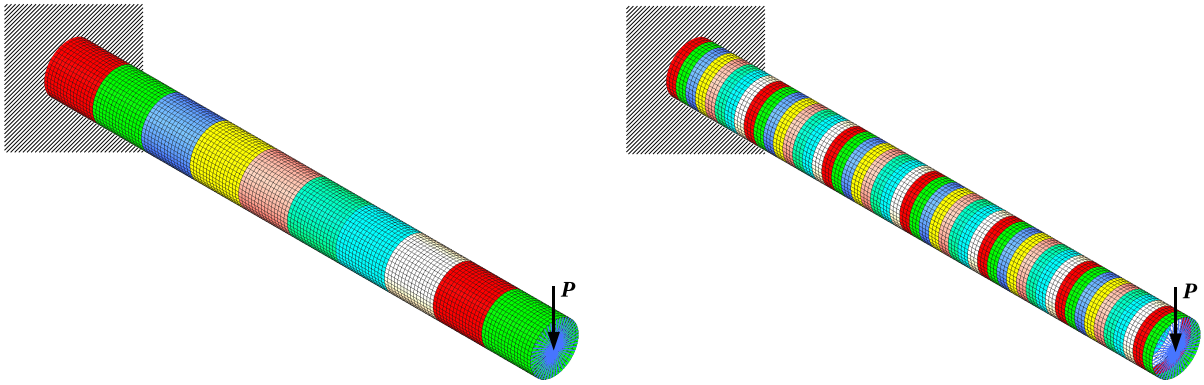
There is also some variation in the number of iterations required for convergence. Both algorithms start with a smaller number of iterations that increases with an increase in problem size, before levelling off at around 16 design variables. For 16 or more design variables DOT required between 12 and 23 iterations, while the new filter-based algorithm required between 18 and 30 iterations. On average, DOT required 14.99 iterations for the 13 problems considered, while the filter-based algorithm required 20.38.

The results generated for this example problem illustrate both the potential accuracy and efficiency of the filter-based algorithm as compared to the traditional SQP algorithm. In terms of accuracy, the objective function values found by the two algorithms correlated very well. In terms of efficiency, the filter-based algorithm do require more iterations than DOT, but never more than twice. Based on the discussion of Section III the filter based algorithm would thus still outperform the standard SQP algorithm in a parallel environment for the test case considered here.

B. Example problem 2: Finite Element Simulation

By comparing the filter-based algorithm against a commercially available algorithm in a serial environment, Section A provided a validation of both the accuracy and potential parallel efficiency of the filter-based algorithm within the context of general-purpose engineering optimization applications. The second example concentrates on the performance

of the filter-based algorithm in a parallel environment. The parallel filter-based algorithm is compared against a parallel implementation of the same commercial SQP algorithm used in Section A. This second example problem consists of a derivation of the first example, where a cantilevered pipe with a hollow circular cross section is considered as shown in Fig. 2. However, unlike the first example, a numerical simulation in the form of a linear finite element analysis is used. Two cases are considered. The first has 10 design variables, representing a small to medium sized general-purpose engineering optimization problem and the second with 50 design variables, representing a medium to large sized general-purpose engineering optimization problem.



(a) 10 Design variable pipe example (b) 50 Design variable pipe example

Figure 2. Pipe example

A steel pipe with material properties E =206 GPa, ν =0.28 and ρ =7620 kg/m^3 is considered. It is assumed that the pipe is completely fixed at the wall (all three displacement and all three rotational degrees of freedom in the finite element model are restrained), while a vertical tip load is applied at the free end. The load is applied at the center point of the cross section, at a node that is connected with rigid elements to the circumference of the pipe. A finite element model that consists of 7 050 four noded linear shell elements and that has 42 306 degrees of freedom is used to analyze the structure. Each finite element analysis calculates the displacement and Von Mises stress values due to the tip load, as well as the natural frequencies of the beam. To increase the computational time of a single analysis (in an attempt to simulate a simple real world finite element analysis), the first 200 frequencies are calculated. As a result, a single analysis takes roughly 50 seconds to complete.

For the optimization problem, the pipe is split into multiple property groups. Each property group consists of a closed ring of elements. This process results in 10 property groups (or rings) for the 10 design variable problem and 50 property groups (or rings) for the 50 design variable problem, as shown in Fig. 2. The objective is to minimize the mass

of the pipe, by changing the thickness value associated with each property group. There is thus a total of either 10 or 50 thickness variables, each with an initial value of 25 mm and each allowed to change between 1 mm and 500 mm. In terms of constraints, a maximum tip displacement of 10 mm and a maximum Von Mises stress of 150 MPa are enforced, while the first natural frequency must be larger than 35 Hz. The location of the maximum stress within each property set does not change during the optimization. As a result, only the maximum stress for each property set is used as a constraint. In total there are thus 1 displacement, 1 frequency and 150 stress constraints.

The focus of this example problem is to investigate the parallel efficiency of the filter-based algorithm and compare that against the parallel efficiency of a commercially available SQP algorithm. Only a single starting point will be considered, but each problem will be solved with both algorithms using different numbers of design variables and different numbers of processors. The influence of the starting point on the performance of the algorithms is accounted for by first comparing only the parallel speedup of each algorithm and second by comparing the time per iteration for different numbers of processors.

For the 10 design variable case, 2, 4, 8 and 11 processors were used. For the 50 design variables case 2, 4, 8, 16, 32 and 51 processors were used. In the implementation presented here, one additional processor is used in each case as the master processor. This processor only manages communication and does not do any computational work. For the number of design variables considered here, the communication is minimal and the master processor can thus potentially also be used to perform computational work. Running two processes on the master processor is referred to as oversubscribing the processor. In the parallel environment used here, there was no easy mechanism for oversubscribing the master processor and instead the contribution of this additional processor is ignored in the results presented here.

The parallel speedup of the algorithms are obtained by dividing the wall clock time required to complete the optimization study using a single processor, by the wall clock time required to complete the study using the prescribed number of processors. The parallel efficiency is then obtained from the parallel speedup, as follows:

$$Efficiency = 100 * \frac{Speedup}{Speedup_{Theoretical}} \tag{14}$$

For an asynchronous implementation the theoretical speedup would simply be the number of processors used in the parallel environment. However, for our synchronous implementation, the theoretical speedup is obtained

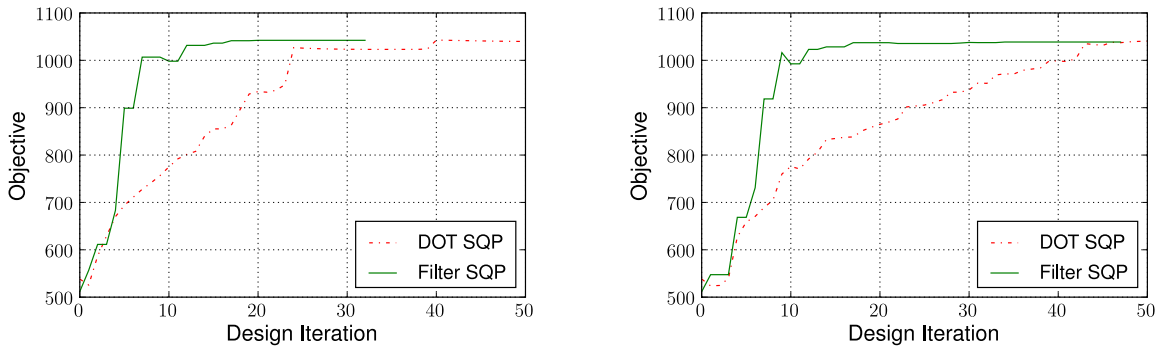
$$Speedup_{Theoretical} = \frac{nTasks}{ceil(\frac{nTasks}{nProc})} \tag{15}$$

where $nTasks$ is the number of analyses to perform in parallel (in our case this is equal to the number of design variables plus 1), *nProc* is the number of processors and *ceil* is the ceiling operator. For example, the 10 variable case requires 11 analyses for each iteration. Since all analyses must be completed before continuing to the next iteration, it makes no difference whether say 6 or 10 processors are used. In both cases there are more processors than half the number of analyses and as a result it would take the equivalent time of two analyses to complete the iteration.

The optimum point found by both algorithms are summarized in Table 2. The filter-based algorithm converged in 35 iterations for the 10 design variable problem and 49 the 50 variable problem. The convergence history in terms of the best objective function value versus the iteration number is shown graphically in Fig. 3. Note that although this is a minimization problem, the objective function values increase in Fig. 3. This increase is due to a highly infeasible starting point.

Table 2. Results for the pipe example problem

Design Variables	Objective		Iterations	
	DOT	Filter	DOT	Filter
10	1039.87	1043.83	50	35
50	1040.30	1040.09	50	49



(a) 10 Design variable problem

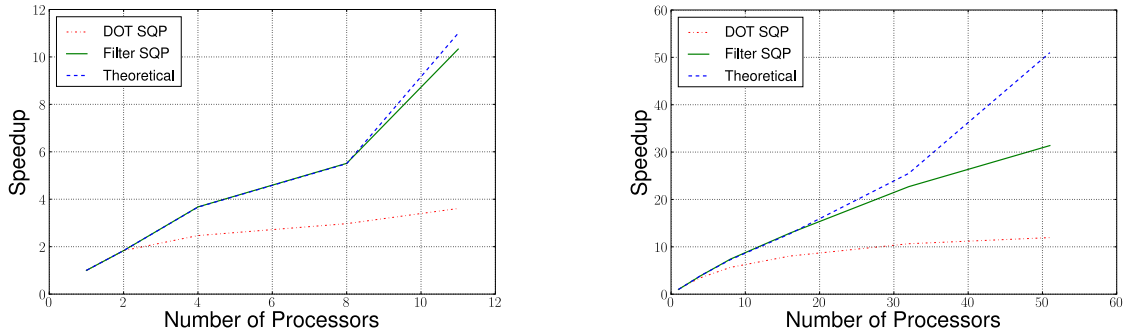
(b) 50 Design variable problem

Figure 3. Iteration history for the pipe example problem

From Fig. 3, it seems that the filter-based algorithm is better suited for the choice of example problem and starting point considered here. For both the 10 and 50 design variable cases, the traditional algorithm had difficulty converging and stopped after 50 iterations, which was the maximum number of iterations allowed for both algorithms. The traditional algorithm had a difficult time satisfying all the constraints (this is an overconstrained problem). The issue could be addressed by tuning algorithm specific parameters or by starting from a different starting point. However, the main goal here was to evaluate the parallel

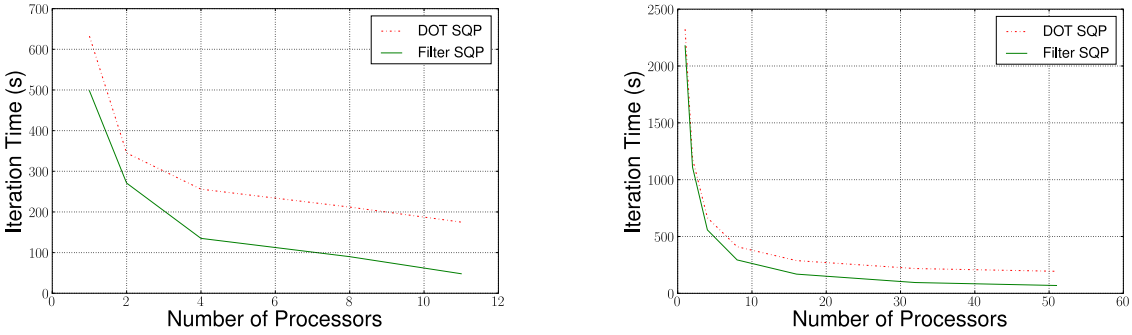
performance of the algorithms and as a result no algorithm specific tuning was performed.

The parallel speedup for both algorithms are shown graphically in Fig. 4, while the average time to complete a single design iteration is shown in Fig.5. In all cases, the execution time for a single processor was estimated from the execution time of the 2 processor case, using Eq. (15). For the 10 design variable case the time was adjusted by a factor of 1.8333, while for the 50 design variable case the time was adjusted by a factor of 1.9615.



(a) 10 Design variable problem (b) 50 Design variable problem

Figure 4. Parallel speedup (theoretical value from Eq. (15))



(a) 10 Design variable problem (b) 50 Design variable problem

Figure 5. Average iteration time

From Fig. 4 it is clear that the filter-based algorithm outperforms the traditional algorithm in terms of parallel speedup. For the 10 design variable case, the filter-based algorithm provides close to theoretical speedup for all number of processors, with a parallel efficiency of 93.9% using 11 processors. In contrast the traditional algorithm had a parallel efficiency of only 32.8% using 11 processors. For the 50 design variables case, the filter-based algorithm provides excellent speedup up to 32 processors with a parallel efficiency of 61.5% when using 52 processors. The traditional algorithm had a parallel efficiency of only 23.4% when using 52 processors.

The reduction in parallel efficiency of the filter-based algorithm with an increase in the number of processors can be explained by considering the effect of the computations performed in serial, on the overall execution time. For the filter-based algorithm, the serial component consists of the time spend in the algorithm itself, mainly to solve the direction finding sub-problem of Eq. (12). In Section A the time spend in the algorithm itself was neglected. However, as more and more processors are used in the parallel environment, the relative contribution of this serial component to the overall execution time increases, with a resulting decrease in parallel efficiency. The increase in relative contribution to the overall execution time, is a result of the computations performed in parallel taking less time, while the time spend on the serial computations remain the same. The influence is bigger for the 50 design variable case, since the direction finding problem is more complex and time consuming than that of the 10 design variable case. The influence of the serial computations on the parallel efficiency will be reduced as the number of design variables are increased (for the same number of processors) or the analysis time of a single analysis increases.

The parallel speedup is also reflected in the time required to complete a single iteration with the filter based algorithm once again out performing the traditional algorithm. In terms of overall execution time, the filter-based algorithm reduced the overall time from 16 930 (using 1 processor) to 1 639 (using 11 processors) seconds for the 10 design variable problem and from 106 741 (using 1 processor) to 3 402 (using 52 processors) seconds for the 50 design variable problem. The traditional algorithm reduced the overall time from 31 624 to 8 759 seconds for the 10 design variable problem and from 116 185 to 9 728 seconds for the 50 design variable problem.

The filter-based algorithm thus has a higher parallel efficiency than the traditional SQP algorithm and on average requires less time to complete a single iteration. Even for a small number of processors, the filter-based algorithm provides a competitive alternative to the traditional SQP algorithm. As the number of processors is increased, the higher parallel efficiency of the filter-based algorithm makes it an easy choice.

VII. Concluding remarks

This paper investigated the use of a filter-based SQP algorithm in a parallel environment. The accuracy and parallel efficiency of the filter-based algorithm were evaluated using two engineering example problems. The first problem was used to establish the accuracy and performance of the algorithm relative to a commercially available algorithm in a serial environment. The second problem was used to evaluate the parallel performance of the new algorithm in a parallel environment.

For the first example problem, the problem size was varied between 2 and 512 variables. The results show that the filter-based SQP algorithm compares very well with a traditional SQP algorithm, both in terms of accuracy and efficiency.

The second example indicates that the algorithm has potential for implementation in a parallel environment. Even with a small number of processors (2, 4 or 8) the new algorithm outperforms the traditional algorithm. As the number of processors is increased, the higher parallel efficiency of the new algorithm makes it an easy choice over the traditional algorithm. For the problems considered here, a drop-off in parallel efficiency was was noticed when using large numbers or processors, relative to the number of design variables. This is to be expected as the relative contribution of the serial computations to the overall execution time is increased with an increase in the number of processors. This effect will be reduced as more design variables are used (for the same number of processors) or the analysis time of a single analysis increases.

Even though the new algorithm clearly outperforms the traditional algorithm in a parallel environment, the parallel performance of the algorithm is influenced by it's synchronous nature, where all analyses within an iteration must be completed before the next iteration is started.

Overall, it seems that the filter-based SQP algorithm is a good candidate for efficiently solving typical engineering problems with up to a few hundred design variables in a parallel environment where the number of processors is less than or equal to the number of design variables plus one. The number of processors is limited by the finite difference gradient calculations. If forward finite differences is used (as was the case here) the maximum number of processors that can be utilized is equal to the number of design variables plus one. The maximum number of processors that can be utilized can be increased by: (1) performing central finite difference gradient calculations, (2) also performing each analysis in parallel, resulting in a two level parallel implementation, and (3) simultaneously starting the algorithm from different starting points to help avoid local minima. The filter-based algorithm provides the efficiency of an established gradient-based algorithm and the parallel efficiency associated with a one step iterative procedure.

Acknowledgments

The authors would like to thank the High Performance Computing Center Stuttgart (HLRS) of the University of Stuttgart for providing access to one of their Linux clusters. Access to this cluster was crucial in obtaining accurate timing information for the parallel benchmarking.

This work has been supported in part by the National Research Foundation (NRF) of South Africa. Any opinion, findings and conclusions or recommendations expressed in this material are those of the author(s) and therefore the NRF does not accept any liability in regard thereto.

References

[1]Venter, G. and Watson, B., "Exploiting Parallelism in General Purpose Optimization," *Proceedings of the 6th International Conference on Applications of High Performance Computing in Engineering, Maui, Hawaii*, Jan 26 – 28 2000, pp. 21–30.

[2]Venter, G. and Sobieszczanski-Sobieski, J., "A Parallel Particle Swarm Optimization Algorithm Accelerated by Asynchronous Evaluations," *Journal of Aerospace Computing, Information, and Communication*, Vol. 3, No. 3, Mar 2006, pp. 123–137.

[3]Koh, B., George, A. D., Haftka, R., and Fregly, B., "Parallel Asynchronous Particle Swarm Optimization," *International Journal for Numerical Methods in Engineering*, Vol. 67, No. 4, 2006, pp. 578–595.

[4]*DOT. Design Optimization Tools, Version 5.x, Users Manual*, Vanderplaats Research and Development, Inc., 1767 S. 8th St., Suite 100, Colorado Springs, CO, January 2001.

[5]Wilson, R., *A Simplicial Algorithm for Concave Programming*, Ph.D. thesis, Harvard University, Graduate School of Business Administration, 1963.

[6]Han, S., "A Globally Convergent Method for Nonlinear Programming," *Journal of Optimization Theory and Applications*, Vol. 22, No. 3, 1977, pp. 297–309.

[7]Powell, M., *A Fast Algorithm for Nonlinearly Constrained Optimizaiton Calculations*, In: Watson, G.A. (ed.), Numerical Analysis, Springer, Berlin, 1978.

[8]Vanderplaats, G. N., *Numerical Optimization Techniques for Engineering Design*, Vanderplaats Research and Development, Inc., 1767 S. 8th St., Suite 100, Colorado Springs, CO, 4th ed., 2005.

[9]Fletcher, R. and Leyffer, S., "Nonlinear Programming without a Penalty Function," *Mathematical Programming*, Vol. 91, No. 2, 2002, pp. 239–269.

[10]Maratos, N., *Exact Penalty Function Algorithms for Finite Dimensioned and Optimization Problems*, Ph.D. thesis, Imperial College of Science and Technology, 1978.

[11]*Open MPI: Open Source High Performance Computing*, http://www.open-mpi.org.